A Universal Infrastructure for the Run-time Monitoring of Parallel and Distributed Applications*

Roland Wismüller, Jörg Trinitis, Thomas Ludwig

Technische Universität München (TUM), Informatik
Lehrstuhl für Rechnertechnik und Rechnerorganisation (LRR-TUM)
Arcisstr. 21, D-80333 München
email: {wismuell|trinitis|ludwig}@in.tum.de

Abstract. On-line tools for parallel and distributed programs require a facility to observe and possibly manipulate the programs' run-time behavior, a so called monitoring system. Currently, most tools use proprietary monitoring techniques that are incompatible to each other and usually apply only to specific target plattforms. The On-line Monitoring Interface Specification (OMIS) is the first specification of a universal interface between different tools and a monitoring system, thus enabling interoperable, portable and uniform tool environments. The paper gives an introduction into the basic concepts of OMIS and presents the design and implementation of an OMIS compliant monitoring system (OCM).

1 Introduction

The development and maintenance of parallel programs is inherently more expensive than that of sequential programs. This is partly due to the lack of widely available, powerful tools supporting test and production phases of those programs, e.g. performance analyzers, debuggers, or load balancers. What these tools have in common is their need for a module that enables them to observe and possibly manipulate the parallel application, a so called *monitoring* module. Usually every tool includes a monitoring module that is specifically adapted to its needs. Since these modules typically use exclusive low level interfaces to the operating system and possibly the underlying hardware, these layers tend to be incompatible to each other. As a result, it is impossible to connect two or more tools to a running application at the same time. Even worse, if you want to use two different tools, one after the other, you often have to re-compile, re-link, or otherwise re-process the application.

In this paper we will describe the design and implementation of a universally usable on-line monitoring infrastructure that is general enough to support a wide variety of tools and their interoperability. We will first describe existing approaches, analyze the requirements for a general monitoring interface, then introduce our approach and finally describe our implementation.

* This work is partly funded by *Deutsche Forschungsgemeinschaft*, Special Research Grant SFB 342, Subproject A1.

2 State of the Art

Existing tools can be classified into interactive vs. automatic, observing vs. manipulating, and on-line vs. off-line tools. Taking a look at existing tools for parallel and distributed programming, one can state that:

- there are more off-line tools than on-line tools,
- most tools are only available on one or very few platforms,
- different on-line tools can typically not be applied concurrently,
- there are no generally available integrated tool environments.

There are several reasons for this. First, developing on-line monitoring systems is complex and expensive. Second, these systems are typically difficult to port from one platform to another. And third, they are usually specifically adapted and restricted to the tools that sit on top of them and require incompatible modifications to the application. Nevertheless, very powerful on-line tools have been developed. Two that are especially interesting concerning their monitoring parts will be mentioned here.

In the area of debugging tools, p2d2 by Cheng and Hood [6] is an important representative. Different from most earlier approaches, p2d2 clearly separates the platform specific on-line monitoring system from portable parts of the tool itself. To accomplish this, a client-server approach is used [7]. The p2d2 server contains all platform specific parts, whereas the client consists of the portable user interface. Client and server communicate via a specified protocol. Cheng and Hood encourage platform vendors to implement debugger servers that comply with the p2d2 protocol. Users could then use an existing p2d2 debugger client to debug applications on different machines.

Taking a look at performance analysis tools, the ambitious Paradyn project by Miller and Hollingsworth [10] clearly deserves a closer look. Paradyn utilizes the exceptional technique of dynamic instrumentation [5], where code snippets performing performance analysis tasks are dynamically inserted into and removed from the running process's code. The interface between the tool and the dynamic instrumentation library *dyninst* is clearly defined and published [3]. On the higher level, Paradyn uses the so called W^3-model to automatically determine performance bottlenecks [4, 10]. The W^3-model in particular profits from dyninst because intrusion can be kept low by removing unnecessary instrumentation.

The above tools are two of the very few that define an explicit interface between the monitoring system and the tool itself. By clearly defining this interface it becomes possible to separate the development of the tool from that of the on-line monitoring system. In addition, porting the tool becomes much easier, thus improving the availability of tools. Unfortunately, p2d2 and Paradyn both target only a single class of tools: p2d2 is limited to debugging, whereas Paradyn/dyninst is concentrated on performance analysis. It is impossible to build new tools or interoperable tools with either of these environments. OMIS, on the other hand, is designed with that primary goal in mind.

3 The OMIS Approach

One key idea in OMIS (*On-line Monitoring Interface Specification*) [9] is to define a standardized interface between the tools and the on-line monitoring system. This is comparable to p2d2 and Paradyn/dyninst. Our approach however is much more general as we do not concentrate on a single class of tools. OMIS supports a wide variety of on-line tools, including centralized, decentralized, automatic, interactive, observing, and manipulating tools (off-line tools can be implemented by a tiny on-line module writing trace files).

The tools we initially had in mind were those in THE TOOL-SET [12], developed at LRR-TUM: VISTOP (program flow visualization), PATOP (performance analysis) [2], DETOP (debugging) [11], CoCheck (checkpointing/process migration), LoMan (load balancing), Codex (controlled deterministic execution), and Viper (computational steering). But we do not only want to support classical stand-alone tools. OMIS is intended to also support interoperable tools (two or more tools applied at the same time), e.g. a performance analyzer covering and measuring the dynamic load balancing and process migration as well. Another example is a debugger interoperating with a checkpointing system and a deterministic execution unit, thus enabling the user to start debugging sessions from checkpoints of long lasting program runs. OMIS will also allow to build uniform tool environments. It will thus become possible to develop portable development environments for parallel and distributed programming.

In order to achieve all of the above, the monitoring system has to be programmable. OMIS supports this by specifying so called event/action-relations. An event/action-relation consists of a (possibly empty) event definition and a list of actions, where the event specification defines a condition for the actions to be invoked. Every time the event occurs, the actions are executed (in case of an empty event, they are executed immediately). The OMIS core already provides a rich set of events and actions. In addition to this, there are user defined events.

The main interface between the tool and the monitoring system consists of only a single function. To this function, requests from the tool are specified as event/action-relations in a machine independent string representation. In order to minimize intrusion, requests can be quickly disabled and re-enabled when e.g. a measurement is unnecessary for a while. When it is no longer needed, the whole request can be deleted from the monitoring system. Depending on how often the corresponding event occurs and what flags were specified, a single request can result in $0...n$ replies while it is active.

Every tool that connects to an OMIS compliant monitor is able to define its own view of the observed system. This is done by explicitly attaching to the nodes and processes of interest. Different tools can thus have different views, and are only informed about events taking place within their observed system. This is especially useful in non-exclusively used target environments.

While not being object oriented in the usual sense, objects form an important part of the interface. OMIS represents processes, threads, nodes, requests etc. as objects, which are identified by so called tokens. To improve scalability, all events and actions can operate on object *sets*. The objects are automatically localized within the observed (distributed) system and the requests are distributed according to the objects' locations (location transparency). OMIS also automatically converts object tokens where appropriate. For example, a process token can be converted into the token of the node where this process is currently located.

Although the above concepts are very powerful if wisely applied, there always remain situations where they do not suffice to support some specific environment. To overcome this, OMIS employs another concept: extendibility. Two types of extensions are defined: *Tool extensions* (e.g. for distributed tools) introduce new actions into the monitoring system, whereas *monitor extensions* can introduce new events, actions and objects. One monitor extension is, for example, the PVM extension. It introduces message buffers, barrier objects and services (events and actions) related to the PVM programming model.

A detailed description of the current version of OMIS can be found in [9] and on the OMIS home page in the WWW[1].

4 Design of an OMIS Compliant Monitoring System

Starting from the OMIS specification, we have done a first implementation of an OMIS compliant monitoring system, called OCM. In contrast to OMIS itself, which is not oriented towards specific programming libraries or hardware platforms, the OCM currently targets PVM applications on networks of UNIX workstations. However, we also plan to extend this implementation to MPI and NT workstations.

Goals of the OCM project. The implementation of the OCM serves three major goals. First, it will be used as an implementation platform for the ongoing tool developments at LRR-TUM, and also at other institutes. E.g. the MTA SZTAKI Research Institute of the Hungarian Academy of Science will use the OCM to monitor programs designed with the GRADE graphical programming environment[8]. [2] The second goal is to validate the concepts and interfaces specified by OMIS and to gain experience guiding further refinements of the specification. Finally, the OCM also serves as a reference for other implementors that want to develop an OMIS compliant monitoring system. Therefore, the OCM will be made available under the terms of the GNU license.

Global structure of OCM. Since the current target platform for the OCM is a distributed system where virtually all of the relevant information can only be retrieved locally on each node, the monitoring system must itself be distributed. Thus, we need one or more local monitoring components on each node. We decided to have one additional process per node, which is automatically started when a tool issues the first service request concerning that node. An alternative would be to completely integrate the monitoring system into the programming

[1] `http://wwwbode.informatik.tu-muenchen.de/~omis`

[2] Funded by the German/Hungarian intergovernmental project GRADE-OMIS, contract UNG-053-96.

libraries used by the monitored application, as it is done with many trace collectors. However, we need an independent process, since an OMIS compliant monitoring system must be able to autonomously react on events, independently of the application's state. In addition, the monitoring support offered by the UNIX operating system requires the use of a separate process.

A principal decision has been to design these monitor processes as independent local servers that do not need any knowledge on global states or the other monitor processes. Thus, each monitor process offers a server interface that is very similar to the OMIS interface, with the exception that it only accepts requests that can be handled completely locally.

However, since OMIS allows to issue requests that relate to multiple nodes, we need an additional component in the OCM. This component, called Node Distribution Unit (NDU), has to analyze each request issued by a tool and must split it into pieces that can be processed locally on the affected nodes. E.g. a tool may issue the request `:thread_stop[p_1,p_2]`[3] in order to stop all threads in the processes identified by the tokens *p_1* and *p_2*. In this case, the NDU must determine the node executing process *p_1* and the node executing *p_2*. If these nodes are different, the request must be split into two separate requests, i.e. `:thread_stop[p_1]` and `:thread_stop[p_2]`, which are sent to the proper nodes. In addition, the NDU must also assemble the partial answers it receives from the local monitor processes into a global reply sent to the tool.

In order to keep our first implementation of the OCM manageable, the NDU is currently implemented as a single central process. Thus, a typical scenario of the OCM looks as in Figure 1. The figure also shows that the NDU contains a request parser that transforms the requests specified by the tools into a preprocessed binary data structure that can be handled efficiently by all other parts of the monitoring system.

The communication between all of these components is handled by a small communication layer offering non-blocking send and interrupt-driven receive operations. We need interrupt-driven receives (or related techniques like Active Messages) to avoid blocking of the monitor processes. It would result in the inability to timely react on occurrences of other important events. A polling scheme using non-blocking receive operations is not feasible, because it consumes too much CPU time and thus leads to an unacceptable probe effect. In the current version of the OCM, the communication layer is based on PVM. However, we are working on an independent layer based on sockets, which is necessary for the MPI port. As is indicated in Figure 1, tools are linked against a library that implements the procedural interface defined by OMIS and transparently handles the connection to the NDU and the necessary message passing.

Structure of the local monitor. The local monitor process operates in an event driven fashion. This means that the process is normally blocked in a system call (either `wait` or `poll`, depending on whether we use `ptrace` or `/proc` for process

[3] The leading colon in the request is the separator between event definition and action list. Since the event definition is empty, the action must be executed immediately.

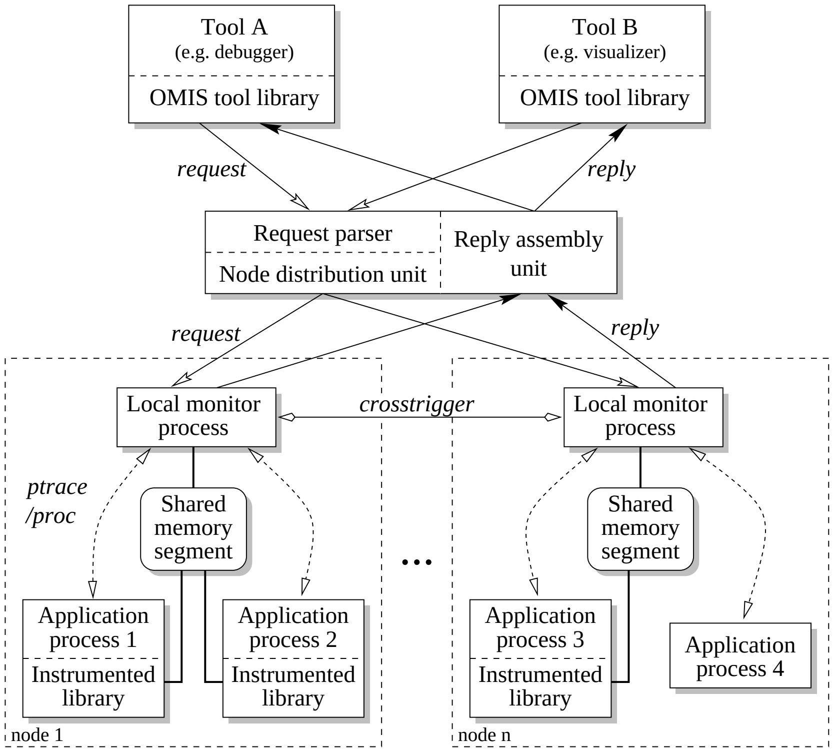

Fig. 1. Coarse Structure of the OCM

control), which returns when an event in one of the monitored processes has been detected by the operating system. In addition, the call is interrupted when a message arrives. After the monitor process has been unblocked, it analyzes the exact kind of event that has occurred and executes the list of actions that are associated with that event. If the event happens to be the arrival of a service request, the associated action causes the request to be received and analyzed. In the case of an unconditional request (i.e. a request with an empty event definition), it is executed and the results are sent back. When a conditional request is received, the monitor process dynamically inserts or activates the necessary instrumentation of the target system and stores the request's action list, so it can be executed later when the event occurs.

Unfortunately, this relatively easy execution scheme is insufficient due to efficiency constraints. For example, assume that a performance analysis tool wants to measure the time process *p_1* spends in `pvm_send` calls. The tool could then issue the following requests:

```
thread_has_started_lib_call([p_1],"pvm_send"): pa_timer_start(ti_1)
thread_has_ended_lib_call([p_1],"pvm_send"): pa_timer_stop(ti_1)
```

Thus, a timer is started when `pvm_send` is called by *p_1*[4] and is stopped again when the call returns. If event detection and action execution are only performed in the monitor process, we introduce an overhead of at least four context switches for each library call, which renders performance analysis useless.

Thus, the OCM must be able both to detect certain events within the context of the triggering application process and to execute simple actions directly in that process, thereby removing the need for context switching. To achieve this, the monitored processes must be linked against instrumented libraries, i.e. extended versions of standard programming libraries that include additional parts of the monitoring system. In order to tell the monitoring parts in these libraries what they shall do, we use a shared memory segment that stores information on which events to monitor and which actions to execute. Note that actions in the application processes should not return results (except in the case of errors), since this would again result in context switches. Rather they update data stored in the shared memory segment (e.g. the timer value in the above example), which are read by actions executed in the monitor process upon request from a tool.

With this extension, the final structure as shown in Figure 1 results. Note that the OCM does not require that each monitored application process is linked with an instrumented library. However, if it is not, some events and actions may not be available for this process.

5 Project Status and Future Work

A first implementation of the OCM, as it is described in Section 4, has already been finished. It is currently supporting PVM 3.3.x and PVM 3.4 on networks of workstations running Solaris, Linux, and Irix. However, some of the services defined by OMIS still need to be implemented. In order to test the OCM, the TOOL-SET debugger has been adapted and is already operational.

Our current work includes the specification of an interface allowing extensions of OMIS compliant monitoring systems to be built independently of the individual implementation of these monitoring systems. This interface is supported by a stub generator that eases the programming of new events and actions.

Future work will focus on adapting the OCM to the MPI implementation `mpich`, and on porting it to the operating system Windows NT. In addition, we will use the OCM as the basis for providing interoperable tools within THE TOOL-SET project[12]. Finally, as a more long term goal, we will implement the OCM also for distributed shared memory systems, both based on hardware support (SCI coupled cluster of PCs using a hardware monitoring module [1]) and pure software implementations.

[4] More exactly: when `pvm_send` is called by *any thread in* process *p_1*. However, since PVM is not multi-threaded, there is only one thread in the process.

6 Acknowledgments

We would like to thank Manfred Geischeder, Michael Uemminghaus, and Hans-Günter Zeller for their participation in the OCM design. Klaus Aehlig, Alexander Fichtner, Carsten Isert, and Andreas Schmidt are still busy with implementing the OCM. We appreciate their support. Special thanks to Michael Oberhuber and Vaidy Sunderam who have been heavily involved in discussing OMIS, and to all other people that provided feedback.

References

1. G. Acher, H. Hellwagner, W. Karl, and M. Leberecht. A PCI-SCI Bridge for Building a PC-Cluster with Distributed Shared Memory. In *Proc. 6th Intl. Workshop on SCI-based High-Performance Low-Cost Computing*, pages 1–8, Santa Clara, CA, Sept. 1996.
2. O. Hansen and J. Krammer. A Scalable Performance Analysis Tool for PowerPC Based MPP Systems. In Mirenkov et al., editors, *The First Aizu Intl. Symp. on Parallel Algorithms / Architecture Synthesis*, pages 78–84, Aizu, Japan, Mar. 1995. IEEE Comp. Soc. Press.
3. J. K. Hollingsworth and B. Buck. *DynInstAPI Programmer's Guide Release 1.1.* Computer Science Dept., Univ. of Maryland, College Park, MD20742, May 1998.
4. J. K. Hollingsworth and B. P. Miller. Dynamic Control of Performance Monitoring on Large Scale Parallel Systems. In *Intl. Conf. on Supercomputing*, Tokio, July 1993.
5. J. K. Hollingsworth, B. P. Miller, and J. Cargille. Dynamic Program Instrumentation for Scalable Performance Tools. In *Scalable High-Performance Computing Conference*, pages 841–850, Knoxville, TN, May 1994.
6. R. Hood. The *p2d2* Project: Building a Portable Distributed Debugger. In *Proc. SPDT'96: SIGMETRICS Symposium on Parallel and Distributed Tools*, pages 127–136, Philadelphia, Pennsylvania, USA, May 1996. ACM Press.
7. R. Hood and D. Cheng. Accomodating Heterogeneity in a Debugger - A Client-Server Approach. In *Proc. 28th Annual Hawaii Intl. Conf. on System Sciences, Volume II*, pages 252–253. IEEE, Jan. 1995.
8. P. Kacsuk, J. Cunha, G. Dzsa, J. Lourenco, T. Antao, and T. Fadgyas. GRADE: A Graphical Development and Debugging Environment for Parallel Programs. *Parallel Computing*, 22(13):1747–1770, Feb. 1997.
9. T. Ludwig, R. Wismüller, V. Sunderam, and A. Bode. *OMIS — On-line Monitoring Interface Specification (Version 2.0)*, volume 9 of *LRR-TUM Research Report Series*. Shaker Verlag, Aachen, Germany, 1997. ISBN 3-8265-3035-7.
10. B. P. Miller, J. M. Cargille, R. B. Irvin, K. Kunchithap, M. D. Callaghan, J. K. Hollingsworth, K. L. Karavanic, and T. Newhall. The Paradyn parallel performance measurement tools. *IEEE Computer*, 11(28), Nov. 1995.
11. M. Oberhuber and R. Wismüller. DETOP - An Interactive Debugger for PowerPC Based Multicomputers. In P. Fritzson and L. Finmo, editors, *Parallel Programming and Applications*, pages 170–183. IOS Press, Amsterdam, May 1995.
12. R. Wismüller, T. Ludwig, A. Bode, R. Borgeest, S. Lamberts, M. Oberhuber, C. Röder, and G. Stellner. The Tool-set Project: Towards an Integrated Tool Environment for Parallel Programming. In *Proc. 2nd Sino-German Workshop on Advanced Parallel Processing Technologies, APPT'97*, Koblenz, Germany, Sept. 1997.